Compilers

Dr. Sherin ElGokhy

Lecture#6

Syntax-Directed Translation
Recursive Descent Parsing

Outline

- Constructing a parse tree
- Recursive descent
  - Algorithms
  - Limitations
- Left Recursion

Abstract Syntax Trees

- So far a parser traces the derivation of a sequence of tokens
- The rest of the compiler needs a structural representation of the program (An actual data structure that tells what the operations in the program and how they are put together)
- Abstract syntax trees
  - Like parse trees but ignore some details
  - Abbreviated as AST

Abstract Syntax Tree. (Cont.)

- Consider the grammar

  $E \rightarrow \text{int} \mid (\ E\ ) \mid E + E$

- And the string

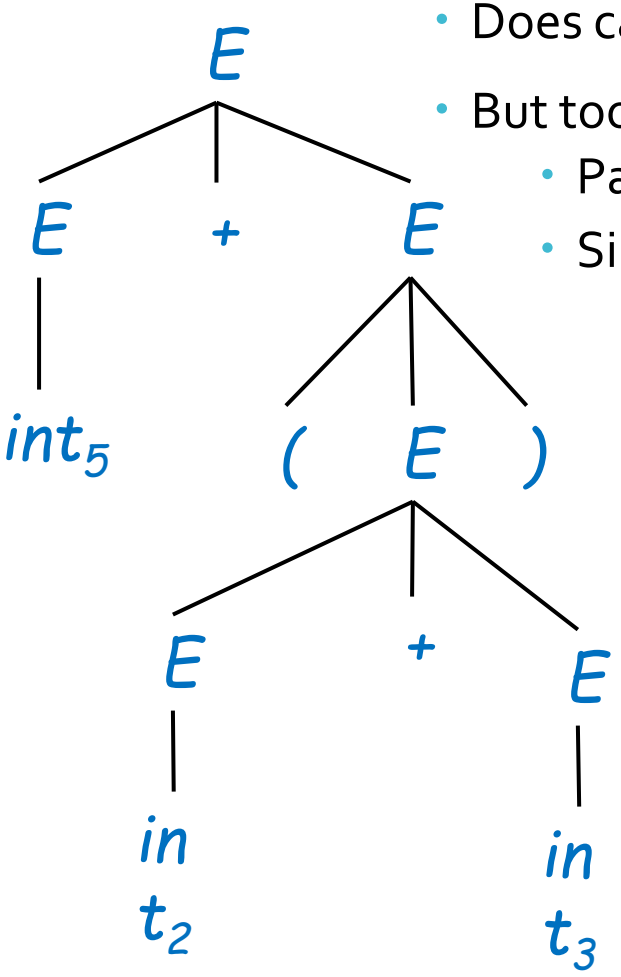
  $5 + (2 + 3)$

- After lexical analysis (a list of tokens)

  int_5 '+' '(' int_2 '+' int_3 ')'

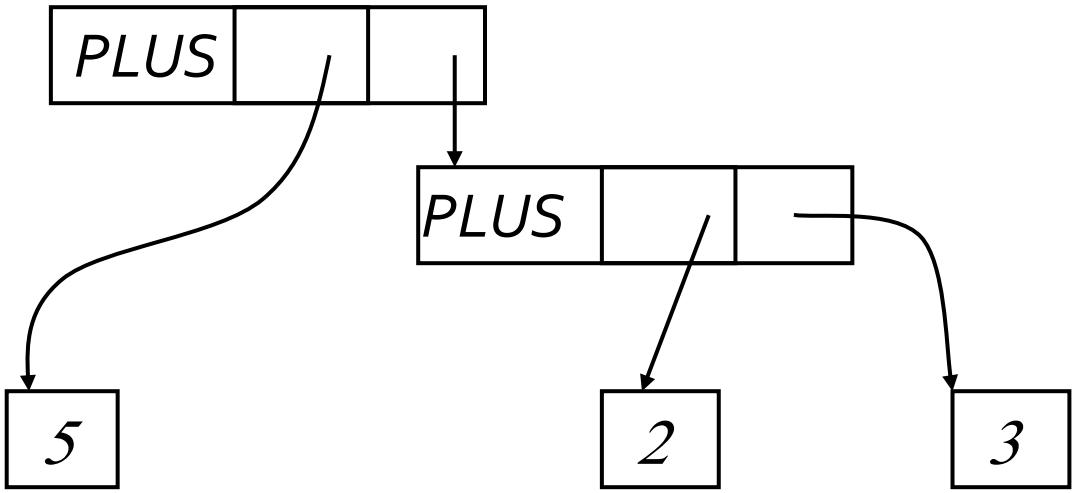
- During parsing we build a parse tree …

Example of Parse Tree



- Traces the operation of the parser
- Does capture the nesting structure
- But too much info
  - Parentheses
  - Single-successor nodes

Example of Abstract Syntax Tree



- AST represent the same thing as the parse tree, but it compress out all the junk in the parse tree…..much simple
- AST also captures the nesting structure
- But abstracts from the concrete syntax
  => more compact and easier to use
- An important data structure in a compiler

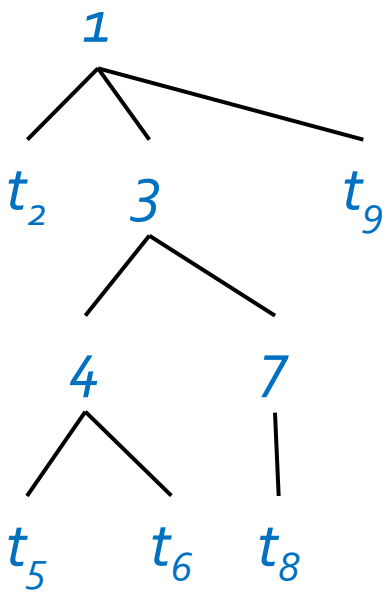
Summary

- We can specify language syntax using CFG
- A parser will answer whether $s \in L(G)$
  - … and will build a parse tree
  - … which will be converted to an AST
  - … and pass on to the rest of the compiler

Intro to Top-Down Parsing: The Idea

- The parse tree is constructed
  - From the top
  - From left to right
- Terminals are seen in order of appearance in the token stream:

  t_2 t_5 t_6 t_8 t_9



Recursive Descent Parsing

- Consider the grammar

  $E \rightarrow T \mid T + E$

  $T \rightarrow int \mid int * T \mid (E)$

- Token stream is: (int_5)

- Start with top-level non-terminal E
  - Try the rules for E in order

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$

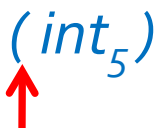
E

(int_5)

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$



Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$

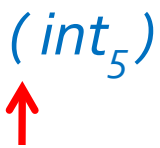
Mismatch: int is not (!
Backtrack …

(int_5)

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

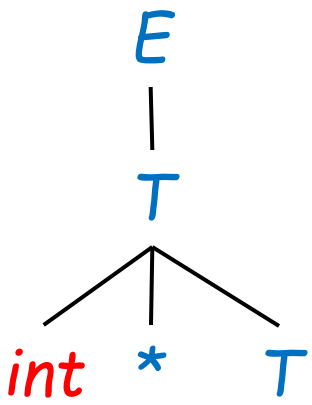
$T \rightarrow int \mid int * T \mid (E)$



Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$



Mismatch: int is not (!
Backtrack …

(int_5)

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

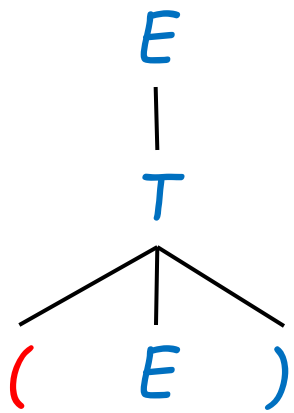
$T \rightarrow int \mid int * T \mid (E)$

(int_5)

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$



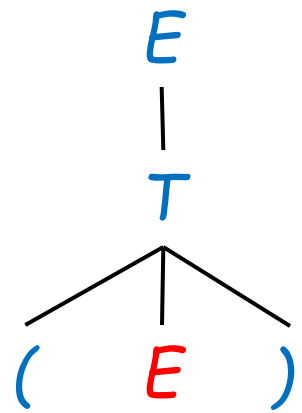
Match! Advance input.

$(\; int_5 \;)$

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$

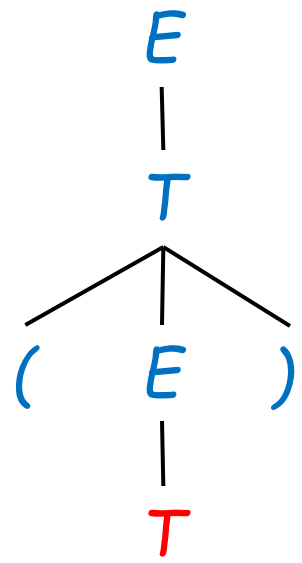


(int_5)

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$

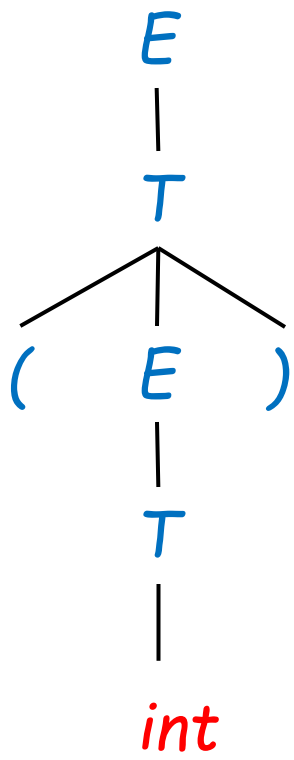


(int_5)

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$



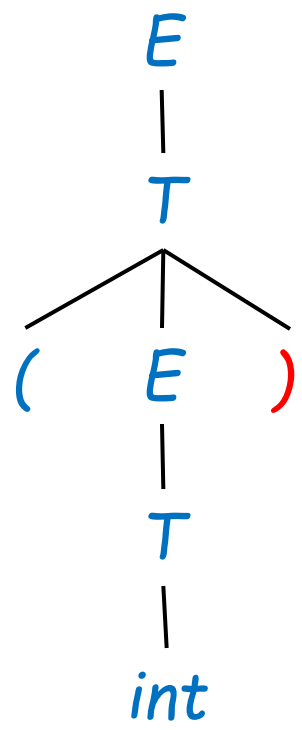
Match! Advance input.

(int_5)

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$



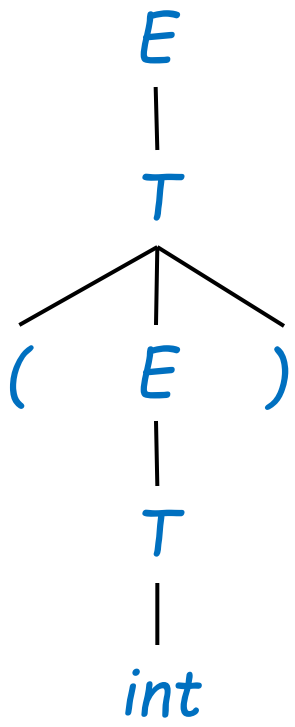
Match! Advance input.

(int_5)

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$



End of input, accept.

(int_5)

Quiz

Choose the derivation that is a valid recursive descent parse for the string id + id in the given grammar. Moves that are followed by backtracking are given in red.

$E \rightarrow E' \mid E' + E$

$E' \rightarrow -E' \mid id \mid (E)$

- E
  E'
  E' + E
  id + E
  id + E'
  id + id

- E
  E'
  -E'
  id
  (E)
  E' + E
  -E' + E
  id + E
  id + E'
  id + -E'
  id + id

- E
  E' + E
  id + E
  id + E'
  id + id

- E
  E'
  id
  E' + E
  id + E
  id + E'
  id + id

A Recursive Descent Parser. Preliminaries

- Let TOKEN be the type of tokens
  - Special tokens INT, OPEN, CLOSE, PLUS, TIMES
- Let the global next point to the next token

A (Limited) Recursive Descent Parser (2)

- Define boolean functions that check the token string for a match of
  - A given token terminal

    bool term(TOKEN tok) { return *next++ == tok; }
  - The n^{th} production of S:

    bool S_n() { … }
  - Try all productions of S:

    bool S() { … }

A (Limited) Recursive Descent Parser (3)

- For production $E \rightarrow T$

  bool E_1() { return T(); }

- For production $E \rightarrow T + E$

  bool E_2() { return T() && term(PLUS) && E(); }

- For all productions of E (with backtracking)

```
bool E() {
  TOKEN *save = next;
  return   (next = save, E1())
        || (next = save,  E2());  }
```

A (Limited) Recursive Descent Parser (4)

- Functions for non-terminal T

```
bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(OPEN) && E() && term(CLOSE); }

    bool T() {
      TOKEN *save = next;
      return    (next = save, T1())
             || (next = save,  T2())
             || (next = save,  T3()); }
```

Example

$E \rightarrow T \mid T + E$ (int)

$T \rightarrow$ int $\mid$ int * T $\mid$ (E)

```
bool term(TOKEN tok) { return *next++ == tok; }

bool E1() { return T(); }

bool E2() { return T() && term(PLUS) && E(); }

bool E() {TOKEN *save = next; return (next = save, E1())
                                  || (next = save, E2()); }

bool T1() { return term(INT); }

bool T2() { return term(INT) && term(TIMES) && T(); }

bool T3() { return term(OPEN) && E() && term(CLOSE); }

bool T() { TOKEN *save = next; return (next = save, T1())
                                  || (next = save, T2())
                                  || (next = save, T3()); }
```

Quiz

Which lines are incorrect in the recursive descent implementation of this grammar?

$E \rightarrow E' \mid E' + id$

$E' \rightarrow -E' \mid id \mid (E)$

- ☐ Line 3
- ☐ Line 5
- ☐ Line 6
- ☐ Line 12

```
1  bool term(TOKEN tok) { return *next++ == tok; }

2  bool E1() { return E'(); }
3  bool E2() { return E'() && term(PLUS) && term(ID); }
4  bool E() {
5      TOKEN *save = next;
6      return (next = save, E1()) && (next = save,  E2());
7  }

8  bool E'1() { return term(MINUS) && E'(); }
9  bool E'2() { return term(ID); }
10 bool E'3() { return term(OPEN) && E() && term(CLOSE); }
11 bool E'() {
12     TOKEN *next = save;  return (next = save, T1())
13                                 || (next = save,  T2())
14                                 || (next = save,  T3());
15 }
```

Recursive Descent Parsing. Notes.

- To start the parser
  - Initialize next to point to first token
  - Invoke E()
- Notice how this simulates the example parse
- Easy to implement by hand

When Recursive Descent Does Not Work…… Limitations

- Try Examples:

int

int*int

- If a production for non-terminal X succeeds, Recursive Descent cannot backtrack to try a different production for X

When Recursive Descent Does Not Work…… Limitations

- Easy to implement by hand
  - But not completely general
  - Cannot backtrack once a production is successful
  - Works for grammars where at most one production can succeed for a non-terminal

When Recursive Descent Does Not Work…… Limitations

- Consider a production $S \rightarrow S\ a$

  bool S_1() { return S() && term(a); }

  bool S() { return S_1(); }

- S() goes into an infinite loop

- A **left-recursive grammar** has a non-terminal S

  $S \rightarrow^+ S\alpha$ for some α

- Recursive descent does not work in such cases

Elimination of Left Recursion

- Consider the left-recursive grammar

$$S \rightarrow S\ \alpha \mid \beta$$

- S generates all strings starting with a β and followed by a number of α

- Can rewrite using right-recursion

$$S \rightarrow \beta\ S'$$

$$S' \rightarrow \alpha\ S' \mid \varepsilon$$

More Elimination of Left-Recursion

- In general

$$S \rightarrow S\,\alpha_1 \mid \ldots \mid S\,\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

- All strings derived from S start with one of $\beta_1,\ldots,\beta_m$ and continue with several instances of $\alpha_1,\ldots,\alpha_n$
- Rewrite as

$$S \rightarrow \beta_1\,S' \mid \ldots \mid \beta_m\,S'$$
$$S' \rightarrow \alpha_1\,S' \mid \ldots \mid \alpha_n\,S' \mid \varepsilon$$

General Left Recursion

- The grammar

$$S \rightarrow A\,\alpha \mid \delta$$

$$A \rightarrow S\,\beta$$

is also left-recursive because

$$S \rightarrow^{+} S\,\beta\,\alpha$$

- This left-recursion can also be eliminated.....**Report Due to Next Lecture**

Quiz

Choose the grammar that correctly eliminates left recursion from the given grammar: $E \rightarrow E + T \mid T$, $T \rightarrow id \mid (E)$

- $E \rightarrow E + id \mid E + (E) \mid id \mid (E)$
- $E \rightarrow TE'$, $E' \rightarrow + TE' \mid \varepsilon$, $T \rightarrow id \mid (E)$
- $E \rightarrow E' + T \mid T$, $E' \rightarrow id \mid (E)$, $T \rightarrow id \mid (E)$
- $E \rightarrow id + E \mid E + T \mid T$, $T \rightarrow id \mid (E)$

Summary of Recursive Descent

- Simple and general parsing strategy
  - Left-recursion must be eliminated first
  - … but that can be done automatically
- Unpopular because of backtracking
  - Thought to be too inefficient
- In practice, backtracking is eliminated by restricting the grammar

Thanks